Case Study: concordance problem

- GIVEN: text (i.e. a sequence of words).
- TO DO: parse the text and discover the words. For each word examine if this its first occurrence.
- If this is the case memorize it.
- Otherwise increment a counter associated to the number of occurrences of the given word.
- Similar issues encountered in analyzing natural language (natural language processing). *What is the text about? Statistical language processing.*
- Also in compilers.
- First phase in a compiler: *lexical analysis.*
- Identifies words. Represents program by *lexical tokens.*
- Example: $x = 5; \text{[IDENT "x"] [EQL] [NUMBER 5] [SEMICOLON]}$.
- To represent $x$: `<identifier>`. Also: pointer to a list (table) of identifiers.
Solution

- Create a linked list of words.
- For each new word:
  - *If not found in the list add it.*
  - *Otherwise increment a counter associated to the given word.*
I am showing only more significant elements in the interfaces;

```cpp
struct Token{
    std::string word;
    int count;
};
struct TokenNode{
    Token info;
    TokenNode * next;
};
class TokenNodeList{
public:
    TokenNodeList();
    void insertOrIncreaseCount (string);
private:
    TokenNode *head;
    TokenNode *tail;
}
```
Comments on implementation

• C++: a `struct` is a class with all members public. It allows e.g. constructors. Though I don’t recommend public data members, it’s a better alternative to structs.

• Useful to maintain list in sorted order: to insert a new word we have to determine anyway that it doesn’t appear in the list.

• Order on words: lexicographic order. The way words are listed in a dictionary: compare first letter first, then second letter, etc. Prefixes are smaller.

• `InsertOrIncreaseCount()`: search until you find element or larger one. If found increase count. Reuse code `insertInfoBefore()`, `insertInfo()` (Course 3).

```c++
int compareStrings(string a, string b){
    int i=0; int l1=a.length(); int l2=b.length();
    for (int i=0;i<min(l1,l2);i++)
        if (a[i]<b[i])
            return 0;
        else
            if (a[i]>b[i])
                return 0;
    return 1;
}
```
void TokenList::insertOrIncreaseCount(string s) {
    assert(!isEmpty());
    node *first = head;
    node *second = 0;
    while (lessThan(first->getInfo(), s)) {
        second = first;
        first = first->getNextNode();
    }
    if (first->getInfo() == s) {
        // string found in list;
        first->count++;
    } else
        insertInfo(second, s);
}
Concordance: main function

```cpp
#include<fstream>

int main(int argc, char *argv[]) {
    std::string nextItem;
    TokenNodeList tl;
    fstream file_op(argv[1],ios::in);

    while (file_op >> nextItem)
        tl.insertOrIncreaseCount(nextItem);
        // do something with token list
        ....
    return 0;
}
```
Skip lists
Skip lists: implementation

- \(k = 1, \ldots, \lceil \log_2(n) \rceil\), \(1 \leq i \leq \lfloor n/2^{k-1} \rfloor - 1\).
- Item \(2^{k-1} \cdot i\) points to item \(2^{k-1} \cdot (i + 1)\).
- every second node points to positions two node ahead,
- every fourth node points to positions four nodes ahead,
- every eigth node points to positions eigth nodes ahead,
- \ldots., and so on.
- Different number of pointers in different nodes in the list!
- half the nodes only one pointer.
- a quarter of the nodes two pointers,
- an eigth of the nodes four pointers,
- \ldots., and so on.
- \(n \log_2(n)/2\) pointers.
- \( n/2 \cdot 1 + n/4 \cdot 2 + n/8 \cdot 4 + \ldots \)
- Each product in the sum is \( n/2 \).
- How many terms in the sum? \( \lceil \log_2 n \rceil \). Total approximately \( n \cdot \frac{\log_2 (n)}{2} \).
- The number of pointers: the level of the node in the tree.
- Levels: from 1 to \( \lfloor \log_2 (n) \rfloor + 1 \).
- To search: first follow pointers on the higher level until a larger element is found or the list is exhausted.
- If a larger element is found, restart search from its predecessor, this time on a lower level.
- Continue doing this until element found, or you reach the first level and a larger element or the end of the list.
find(element el) {
    p = the nonnull list on the highest level i;
    while (el not found and i ≥ 0)
        if (p->key > el)
            p = a sublist that begins in the predecessor of p
            on the level − i;
        else
            if (p->key < el)
                if p is the last element on the level i
                    p = a nonnull sublist that begins in p
                    on the highest level < i;
                i = the number of this level;
            else
                p = p->next;
    }

Inserting and deleting nodes

- Problem: when inserting/deleting a node pointers of following nodes have to be restructured.
- Solution: rather than equal spacing, random spacing on a level.
- Number of nodes on each level approximately preserved.
- Level numbering: start with zero.
- New node inserted: probability 1/2 on first level, 1/4 second level, 1/8 third level, . . . , etc.
- Function \textit{chooseLevel}: chooses the level of the new node.
- Generate random number. If in [0,1/2] level 1, [1/2,3/4] level 2, etc.
- Construct for "typical" case.
- Use randomness to simplify constructions.
#define MAXLEVEL 4

class SkipListNode{
public:
    SkipListNode(){
        int key;
        SkipListNode ** next;
    }

class SkipList{
public:
    SkipList();
    bool isEmpty() const;
    void choosePowers();
    int chooseLevel();
    int * SkipListSearch(int);
    void SkipListInsert(int);
private:
    typedef SkipListNode* nodePtr;
    nodePtr root[MAXLEVEL];
    int powers[MAXLEVEL];
};
SkipList::SkipList(){
    for(int i=0;i<maxLevel;i++)
        root[i]=0;
}

bool SkipList::isEmpty() const{
    return root[0]==0;
}

void SkipList::choosePowers(){
    powers[maxLevel-1]= (int) pow(2,maxLevel-1)-1;
    for(int i= maxLevel-2,j=0;i>=0;i--,j++)
        powers[i]= (int) pow(2,j+1);
}

int SkipList::chooseLevel(){
    int i,r=rand() % powers[maxLevel-1] + 1;
    for(i=1;i<maxLevel;i++)
        if (r< powers[i])
            return i-1; //return a level < than the highest level.
    return i-1; // return the highest level.
nodePtr* SkipList::SkipListSearch(int key){
    if (isEmpty()) return 0;
    nodePtr prev, curr;
    int lvl; // find the highest nonnull
    for (lvl=maxLevel-1; lvl >= 0 && !root[lvl]; lvl--)// level
        prev = curr = root[lvl];
    while (true) {
        if (key == curr->key) // success if equal
            return curr;
        else if (key < curr->key) { // if smaller, go down
            if (lvl == 0) // if possible
                return 0;
            else if (curr == root[lvl])
                curr = * (prev->next + --lvl); // starting from predecessor
            else curr = *(prev->next + --lvl); // starting from predecessor
        } // which can be the root
        else { // greater
else { // greater
    prev = curr;
    if (*(curr->next+lvl) != 0) // go to the next nonnull node
        curr = *(curr->next+lvl); // on the same level
    else { // or a list on a lower level
        for(lvl--; lvl>=0 & & *(curr->next + lvl)==0;lvl--)
            if (lvl == 0)
                curr = *(curr->next+lvl);
        else return 0;
    }
}

void SkipList::skipListInsert(int key){
    nodePtr curr[maxLevel], prev[maxLevel], newNode;
    int lvl, i;
    curr[maxLevel-1]=root[maxLevel-1];
    prev[maxLevel-1]=0;
for(lvl=maxLevel-1;lvl >=0;lvl--){
    while (curr[lvl] &&curr[lvl]->key < key){
        prev[lvl] = curr[lvl];
    }
    if(curr[lvl] && curr[lvl]->key == key)// don’t include
        return; // duplicates
    if (lvl > 0) // go one level down
        if (prev[lvl]==0){ // if not the lowest level,
            curr[lvl-1]=root[lvl-1]; // using a link
            prev[lvl-1]=0; // either from the root
        }
        else { // or from predecessor;
            curr[lvl-1]=*(prev[lvl]->next+lvl-1);
            prev[lvl-1]=prev[lvl];
        }
}
lvl = chooseLevel(); // generate randomly level for node
newNode = new SkipListNode;
newNode->next = new nodePtr[sizeof(nodePtr)* (lvl+1)];
newNode->key = key;
for (i=0; i<= lvl;i++) { // initialize next fields of
    *(newNode->next +i) = curr[i]; // newNode and reset to newNode
    if (prev[i] == 0) // either fields of the root
        root[i] = newNode; // or next fields of newNode’s
    else *(prev[i]->next +i) = newNode; // predecessors
}
Self-organizing lists

- Singly/doubly linked lists: sequential search to find element.
- If we know what is likely to be searched next one can improve efficiency of search by dynamically organizing the list. Depends on the stream of data.
- Move to front: after desired element located, put it at the beginning of the list. Assume that recently seen elements are more likely to be searched again.
- Transpose method: after desired element located, transpose it with previous element (unless at the head).
- Count method: Order the list by the number of times elements are accessed.
- Ordering method: order the list using criteria natural for the information under scrutiny (alphabetical).
- First three methods: new information added at the end. Last one: sorted order.
Comparing efficiency of ordering methods

- Metric: number of comparisons/maximum number of comparisons.
- Usually: with optimal static ordering. All data ordered by frequency of occurrence. Requires two passes.
- Example: text; sum of combined lengths is 46. MTF: 33 comparisons (71.7%), plain search 30(62.5%).
- four benchmarks. second, fourth: computer programs. rest: English language text.
- Count, MTF twice as costly as optimal static ordering.
- of course, table only comparison, not other operations (e.g. pointer movement).

<table>
<thead>
<tr>
<th></th>
<th>Different</th>
<th>All</th>
<th>Optimal</th>
<th>Plain</th>
<th>MTF</th>
<th>Trs.</th>
<th>Count</th>
<th>Alph.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>156</td>
<td>149</td>
<td>609</td>
<td>550</td>
<td>1,163</td>
<td>2,013</td>
<td>61.3</td>
<td>50.9</td>
</tr>
<tr>
<td></td>
<td>347</td>
<td>423</td>
<td>1,510</td>
<td>2,847</td>
<td>5,866</td>
<td>23,065</td>
<td>26.4</td>
<td>45.6</td>
</tr>
<tr>
<td>Optimal</td>
<td>28.5</td>
<td>24.5</td>
<td>17.6</td>
<td>16.2</td>
<td>10.0</td>
<td>18.4</td>
<td>35.4</td>
<td>50.0</td>
</tr>
<tr>
<td>Plain</td>
<td>70.3</td>
<td>67.1</td>
<td>56.3</td>
<td>51.7</td>
<td>35.4</td>
<td>32.9</td>
<td>19.8</td>
<td></td>
</tr>
<tr>
<td>MTF</td>
<td>61.3</td>
<td>54.5</td>
<td>31.3</td>
<td>30.5</td>
<td>18.4</td>
<td>49.4</td>
<td>32.9</td>
<td></td>
</tr>
<tr>
<td>Trs.</td>
<td>68.8</td>
<td>66.1</td>
<td>53.3</td>
<td>49.4</td>
<td>32.9</td>
<td>32.9</td>
<td>19.8</td>
<td></td>
</tr>
<tr>
<td>Count</td>
<td>61.2</td>
<td>54.7</td>
<td>34.0</td>
<td>32.0</td>
<td>19.8</td>
<td>19.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alph.</td>
<td>50.9</td>
<td>48.0</td>
<td>55.7</td>
<td>50.4</td>
<td>50.0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Binary trees

- A tree consists of *nodes* and *arcs*.
- Depicted upside down with its root at the top and the leaves at the bottom.
- Can be defined as follows:
  - An empty structure is a tree.
  - If \( t_1, \ldots, t_k \) are disjoint trees, then the structure whose root has as its children the roots of \( t_1, \ldots, t_k \) is also a tree.
- Only structures generated by rules 1 and 2 are trees.
Examples of trees

(a) (b) (c)
Examples of trees (II)

University

Campus A
- Dept1
  - Prof.
- Dept2
- Dept3
  - Students

Campus B
- Dept1
- Dept2
- Dept3
  - Students
Linked lists vs. trees

Data Structures – p.23/39
Linked lists vs. trees

• Linked lists: search can take $\Omega(n)$ time. For large lists (and frequent search) this is significant.

• If transformed to *orderly* trees (trees with well-defined ordering criterion): can reduce this complexity.

• All trees discussed in this chapter: **binary trees**.

• Level of a node: 1 plus number of edges from root.

• Complete binary tree: $2^i$ nodes at level $i + 1$. 
Binary search trees

- satisfy the following property:
- all nodes in a left subtree smaller than the root node.
- all nodes in right subtree larger.
- duplicate keys: avoided. Treated as *logical error*.
- Node keys: *scalar type* (allows comparisons).
Implementing binary search trees with vectors

- node is a structure with one information field and two "pointer" fields.
- pointer fields: indices of left and right subtrees (if they exist).
- can be inconvenient even for flexible arrays (vectors).
- "holes" in tree need to be eliminated. Populate array with special symbol (-1).
- also need to deal with cycles.
Example: BST with vector

<table>
<thead>
<tr>
<th>Index</th>
<th>Info</th>
<th>Left</th>
<th>Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>{0}</td>
<td>13</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>{1}</td>
<td>31</td>
<td>6</td>
<td>-1</td>
</tr>
<tr>
<td>{2}</td>
<td>25</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>{3}</td>
<td>12</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>{4}</td>
<td>10</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>{5}</td>
<td>2</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>{6}</td>
<td>29</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>{7}</td>
<td>20</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>
Implementing BST with pointers

- class BSTNode contains three fields:
  - integer info,
  - pointers left and right to left and right child.
  - for easier traversal: also contain pointer to parent.
  - all info: handled by class BST, which contains pointer to root.
- thus: members of BST node are public.
class BSTNode{
public:
    BSTNode(){
        left = right = 0;
    }
    BSTNode(int el, BSTNode *l=0, BSTNode *r=0){
        key = el; left = l; right = r;
    }
    int key;
    BSTNode *left, *right;
};
class BST{
public:
BST(){
    root = 0;
}
~BST(){
    clear();
}
void clear(){
    clear(root); root=0;
}
bool isEmpty() const {
    return root == 0;
}
void preorder()
{
    preorder(root);
}

void inorder()
{
    inorder(root);
}

void postorder()
{
    postorder(root);
}

int * search(int el) const
{
    return search(root,el);
}

void breadthFirst();
void iterativePreorder();
void iterativeInorder();
void iterativePostorder();
void iterativePostorder();
void MorrisInorder();
void insert(int);
void deleteByMerging(BSTNode * &);
void findAnddeleteByMerging(int);
void deleteByCopying(BSTNode * &);

protected:
    BSTNode * root;
    void clear(BSTNode *);
    int *search(BSTNode *, int) const;
    void preorder (BSTNode *);
    void inorder (BSTNode *);
    void postorder (BSTNode *);

};
BSTNode * BST::search(BSTNode *p, int el){
    while (p != 0)
        if (el == p->key)
            return p;
        else
            if (el < p->key)
                p = p->left;
            else
                p = p->right;
    return 0;
}
Tree traversals

- Systematic visiting of all nodes in the tree.
- Each node visited once, does not specify order.
- Breadth-first traversal: visit each node, starting from the lowest level and moving down by level, visiting nodes from left to right.
- Depth-first traversals: go in subtree as deep as you can, backtrack. Differ on the order of visiting root, left, right subtrees.
  - preorder: VLR.
  - inorder: LVR.
  - postorder: LRV.
- these definitions are recursive.
void BST::breadthFirst()
{
    Queue<BSTNode> q;
    BSTNode *p = root;
    if (p != 0){
        q.enqueue(p);
        while (!q.empty()){
            p = q.dequeue();
            visit(p);
            if (p->left != 0)
                q.enqueue(p->left);
            if (p->right != 0)
                q.enqueue(p->right);
        }
    }
}
void BST::inorder(BSTNode *p){
    if (p!=0){
        inorder(p->left);
        visit(p);
        inorder(p->right);
    }
}

void BST::preorder(BSTNode *p){
    if (p!=0){
        visit(p);
        preorder(p->left);
        preorder(p->right);
    }
}

void BST::postorder(BSTNode *p){
    if (p!=0){

Preorder traversal
Inorder traversal
Postorder traversal